RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

Topic 3: Abstract Classes and Interfaces

Lecture Slides

What is an `abstract` class?

Simply put, an abstract class is a Java class **that has not been fully defined.** For example, it may have the implementation of a method missing.

This can be useful, as it allows the subclasses to inherit and override these abstract methods; and it gives each of these subclasses **a consistent programming interface.**

Furthermore, the “gap” that the abstract class fills in the class hierarchy **can be used as a declaration type** to handle any extension of the abstract class.

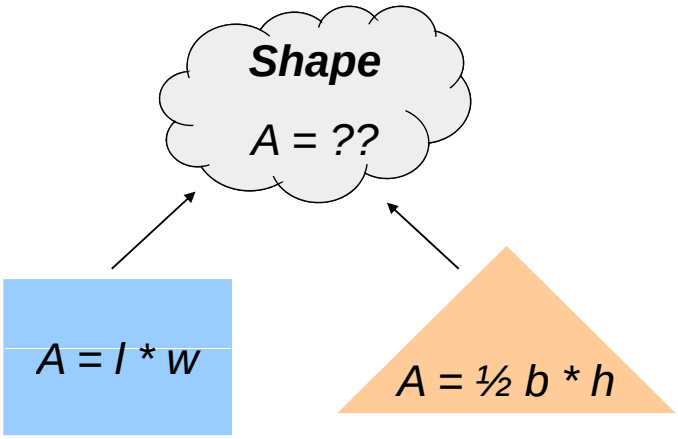
Note: a **fully implemented** (normal) class is referred to as a **concrete class.**

Abstract Classes: an Example

Take the example of a series of shapes, that is to be modelled in a Java class hierarchy.

All shapes will have a colour, and other common elements, however (for example) the manner in which the area of the shape is calculated would depend on what kind of shape it is.

Therefore, it would make conceptual sense to give the area method an implementation only in the concrete subclasses; we can make it abstract to ensure this.



Implementing an Abstract Class

An abstract class (or an abstract method) is defined using the keyword **`abstract`**. It is also possible to define an abstract class that has no abstract properties!

```
abstract class Shape {
  protected Color col;

  public abstract double area();
}
```

Note the lack of an implementation for the **`area()`** method, and the need to declare the class as abstract.

Extending an Abstract Class

An extension of an abstract class only becomes concrete if there are no abstract methods left (otherwise it must itself be declared abstract). In our `Rectangle` class we must therefore override the **`area()`** method:

```
class Rectangle extends Shape {
  double length, width;

  public double area() {
    return (length * width);
  }
}
```


Using Abstract Classes

If there were any `abstract` properties left in our subclass of `Shape`, that subclass could be defined as `abstract`, and a compiler error avoided.

However, it is not possible to make an instance of (instantiate using new) an abstract class. In order to make instances of a class, it must be fully defined i.e. Be a non-abstract concrete class.

```
Shape shape = new Shape();           // this will not work
Rectangle shape2 = new Rectangle();  // this is allowed
```

With the class hierarchy having the abstract **Shape** as its parent, it allows properties common to all Shapes (such as the colour) to be defined and inherited, thus re-using code as much as possible; but it still allows us to leave some implementation details for further down the class hierarchy.

Abstract Classes and Polymorphism (1)

Although abstract classes lack some implementation details, they can still be used in polymorphic behaviour.

```
Shape shape = new Rectangle();
shape.length = 2.5;
shape.width = 5.0;

System.out.println(shape.area());
```

Note that since `shape` is of the static (compile-time) type **`Shape`** we can only call methods declared inside **`Shape`** not specialised methods in **`Rectangle`**.

Abstract Classes and Polymorphism (2)

Here is another example; it is a method that will take any **`Shape`** object, and print its area.

```
public void printArea(Shape in) {
  System.out.println("Area: " + in.area());
}
```

Recall earlier that we could not directly **create** an object from the abstract **`Shape`** class. However in these cases, we are working on concrete subclasses of **`Shape`**; we are just using the **`Shape`** data type as a handle to refer to any subclass.

Remember the ‘is-a’ relationship: *“a **`Rectangle`** is a **`Shape`**”*

What is an interface?

An interface is a **definition of a programming contract** that classes can **implement**. It is **not** a class and cannot be instantiated.

It is possible for a class to implement many interfaces.

The major benefit of using an interface is that it allows otherwise unrelated classes to be given a common **type** without the need for multiple inheritance (which Java forbids.)

If a class implements a particular interface, it must **directly implement** all of the methods of that interface or it must be declared abstract.

If a class that implements an interface is extended, **those subclasses also implement the interface**. (This is because the methods that make the implementation will be inherited.)

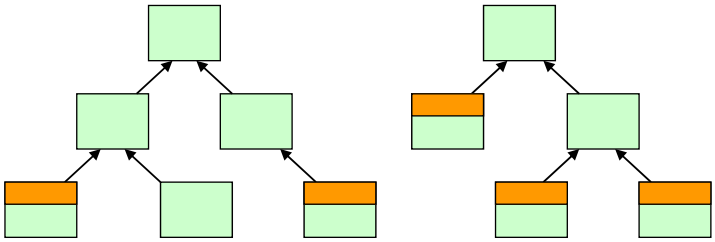
Interfaces: an Example

Consider a set of class hierarchies, some of whose objects can be printed out to the screen.

We could **create an interface** defining how the “printable” functionality is to be implemented, and have these classes implement that interface.

The benefit is that we can then use that interface **as a data type**, much like we can with superclasses and polymorphism.

All `Printable` classes in the diagram can be referred to by their capability, even though they are completely unrelated in the hierarchy.



Defining an Interface

The code to define an `interface` is somewhat similar to an `abstract` class; no method implementations are given, and use of the **`abstract`** keyword is optional. Again, remember that an interface is not a class and cannot be instantiated.

```
interface Printable {
  public void print();
}
```

We can also extend an interface to make a larger sub-interface. This is done using the **`extends`** keyword, much like extending a class. The new interface has the methods of the interface it extends plus any new methods it declares.

Implementing an Interface

In order to get a class to implement the interface, we use the **`implements`** keyword:

```
class StringData implements Printable {
  private String data;

  public void print() {
    System.out.println("Data: " + data);
  }
}
```

We **must** either provide an implementation for **`print()`** or declare the StringData class abstract.

An Interface as a Data Type

As with abstract classes, we can use our **`Printable`** interface as a data type:

```
public void printObject(Printable in) {
  in.print();
}
```

This method will accept any object that implements **`Printable`**, and call its **`print()`** method.

However, when referring to the objects as **`Printable`**, we can only call what is defined inside the **`Printable`** interface.

Using interfaces allows us to write code that can operate on a wide range of specialised data types (by using interface types to declare variables, as parameter types etc.)

An Example: the Shape class

Let's write a more formal example of abstract classes and interfaces.

Specification: define a set of classes to process geometric shapes. There is no basic shape from which all shapes derive, so we place an abstract class at the root of the hierarchy.

```
abstract class Shape {
  public abstract double area();
  public abstract double perimeter(); // yes this is ok!
}
```

The next step is to extend Shape to handle circles and rectangles, and then create instances of these classes:

```
Shape aRect = new Rectangle();
Shape aCircle = new Circle(3.0);
```

An Example: the `Circle` class

```
class Circle extends Shape {
  protected double r;

  Circle(double r) {
    this.r = r;
  }

  public double area() {
    return Math.PI * r * r;
  }

  public double perimeter() {
    return 2 * Math.PI * r;
  }
}
```


An Example: the `Rectangle` class

```
class Rectangle extends Shape
{
   protected double w, h;

   Rectangle(double w, double h)
   {
      this.w = w;    this.h = h;
   }

   public double area()
   {
      return w * h;
   }

   public double perimeter()
   {
      return 2 * (w + h);
   }
}
```

Introducing Interfaces

Suppose we want to derive a subclass of `Rectangle` that allows a rectangle to be drawn on a window object. We might also want to do this with circles as well. What can we do?

Rewrite the Shape hierarchy to include drawing methods

We would override these in each non-abstract shape.

But what if not all shapes are Drawable?

Define drawing methods in each non-abstract drawable shape?

Loses polymorphic behaviour

Can we define a class that extends `Shape` but also has a 'parent' of sorts, that provides methods for drawing shapes in a window object? Java does not allow multiple inheritance, but we can use an `interface` for this job.

```
interface Drawable {
  public void setColor(Color c);
  public void setPosition(double x, double y);
}
```


The `DrawableRectangle` class

```
class DrawableRectangle extends Rectangle implements Drawable {

   private Color c;
   private double x, y;

   DrawableRectangle(double w, double h) {
     super.w = w;
     super.h = h;
   }

   public void setColor(Color c) {
     this.c = c;
   }

   public void setPosition(double x, double y) {
     this.x = x;
     this.y = y;
   }
}
```

Design with Abstract Classes and Interfaces

Why abstract classes?

We want a generalised base class that is incomplete (and thus cannot be accidentally instantiated) but want to provide default behaviour (that may be extended or specialised later).

Why interfaces?

- We want to treat different classes together in some specific way by means of a strict contractual interface (set of method signatures) but have no default behaviour
- We have many such behaviours and therefore the need for multiple inheritance which is only supported at the interface level in Java.

Note that like regular inheritance abstract classes and interfaces ensure a uniform and structured design throughout the class hierarchy. Abstract classes, inheritance and interfaces, together with polymorphism and dynamic binding, promote code reuse by allowing methods to be called in a generalised way. It also facilitates maintainability by placing specialised code in a single place.

Abstract Classes vs. Interfaces

A side-by-side comparison of the two:

Abstract Class	Interface
1. Used by extending.	1. Used by implementing.
2. Can have `abstract` and concrete methods.	2. Can have only `abstract` methods.
3. Need not implement all abstract methods.	3. No methods are implemented.
4. Can have any modifiers on data types.	4. Data must be `public static final` constants.
5. A class cannot extend multiple abstract classes.	5. A class can implement many interfaces.
6. Cannot be instantiated.	6. Cannot be instantiated.
7. Can be used as a data type.	7. Can be used as a data type.

Additional Lecture Resources

PowerPoint slides from chapter 10 Abstract Classes and Interfaces of the prescribed textbook *Introduction to Java Programming by Y. Daniel Liang, Prentice-Hall, 2007*, are available on Blackboard and will be presented during the lecture. You should study the book chapter and slides and may wish to print the slides so that you have a hard copy during the lecture.